多接入边缘计算中相关性任务的联合调度算法

鲁蔚锋[1,2]，李宁[1,2]，徐佳[1,2]，徐力杰[1,2]，徐建[3]

（1. 南京邮电大学计算机学院、软件学院、网络空间安全学院，江苏 南京 210023；
2. 南京邮电大学江苏省大数据安全与智能处理重点实验室，江苏 南京 210023；
3. 南京理工大学计算机科学与工程学院，江苏 南京 210094）

摘 要：多接入边缘计算已经成为资源密集型应用程序的有前途的计算范式。不过，先前大部分研究工作没有考虑到任务的相关性，这可能导致不可行的调度决策。考虑应用程序上有些任务必须要在本地完成，研究了相关性任务在本地和边缘侧的联合调度方法，并考虑了多接入边缘计算卸载场景下的另一个不可忽视的能耗问题。将问题形式化为在满足应用程序的完成截止时间约束的条件下最小化系统中的能耗，并提出联合调度（JS）算法解决该问题。最后通过仿真实验分析 JS 算法在应用程序的完成率和系统能耗两方面的性能。仿真结果表明，JS 算法在应用程序的完成率上优于其他对比算法并且至少可以节省 43%的系统能耗。

关键词：多接入边缘计算；相关性任务；能耗；任务调度；联合调度算法

中图分类号：TN92

文献标志码：A

DOI: 10.11959/j.issn.1000−436x.2023047

Joint scheduling algorithm for correlative tasks in multi-access edge computing

LU Weifeng[1,2], LI Ning[1,2], XU Jia[1,2], XU Lijie[1,2], XU Jian[3]

1. School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing 210023, China
2. Jiangsu Key Laboratory of Big Data Security and Intelligent Processing, Nanjing University of Posts and Telecommunications, Nanjing 210023, China
3. School of Computer Science and Engineering, Nanjing University of Science and Technology, Nanjing 210094, China

Abstract: Multi-access edge computing (MEC) has emerged as a promising computing paradigm for resource-intensive applications. However, most of the previous research work has not considered correlative tasks, which may lead to infeasible scheduling decisions. Considering that some tasks on the application must be completed locally and another non-negligible energy consumption problem in the multi-access edge computing offloading scenario, the joint scheduling algorithm of correlative tasks on the local and edge sides was studied. The problem was formalized as minimizing the energy consumption in the system while satisfying the application's completion deadline constraints, and the joint scheduling (JS) algorithm was proposed to solve the problem. Finally, the performance of the JS algorithm in the application completion rate and system energy consumption were analyzed through simulation experiments. The simulation results show that the JS algorithm is superior to other comparison algorithms in the application completion rate and can save at least 43% of the system energy consumption.

Keywords: multi-access edge computing, correlative task, energy consumption, task scheduling, joint scheduling algorithm

---

收稿日期：2022−09−16；**修回日期：**2023−01−10

通信作者：徐佳，xujia@njupt.edu.cn

基金项目：国家自然科学基金资助项目（No.61872193, No.61971235, No.62072254）

Foundation Item: The National Natural Science Foundation of China (No.61872193, No.61971235, No.62072254)

0 引言

物联网的普及和移动智能设备的广泛使用极大地促进了时延敏感型和资源密集型的应用程序的发展，例如虚拟/增强现实、人脸识别和数据流处理[1-2]。通常，这些应用程序在移动设备或云服务器上执行。但是，移动设备通常不会有足够的计算资源用于资源密集型应用[3-4]。另一方面，在云服务器上运行应用程序通常需要将大量数据从移动设备传输到云中的远程服务器，导致通信时延很长，这对时延敏感型的应用程序非常不友好。因此，多接入边缘计算（MEC, multi-access edge computing）作为一种有前途的计算范式出现在本地计算模式和云计算模式[5-6]。

MEC网络中的边缘节点与云端相比具有有限的计算能力和存储能力，为了更好地优化边缘节点的资源利用率，对任务的卸载与调度策略的研究具有十分现实的意义。近年来，MEC中的任务卸载和调度问题一直是研究工作的重点[7-8]。例如，文献[9]研究位于多个边缘服务器之间的工作负载平衡，提出了一种将任务从过载的边缘服务器卸载到其他负载不足的边缘服务器上执行的调度算法，目标是最小化完成任务的时间。但是，上述文献无法处理相关性任务的调度问题。例如，阿里巴巴的集群跟踪程序[10]，超过75%阿里巴巴数据追踪中的400万个作业（应用）涉及相关任务。图1演示了来自Facebook[11]的视频处理应用程序，其中视频分类计算可以分为多个相关任务。具体来说，任务的执行应该满足各种优先约束，即一个任务不能在其所有前驱任务完成之前开始。此外，跨服务器数据传输通常会发生，因为相关任务安排在不同的边缘节点上，并且这些相关任务之间存在数据依赖性。

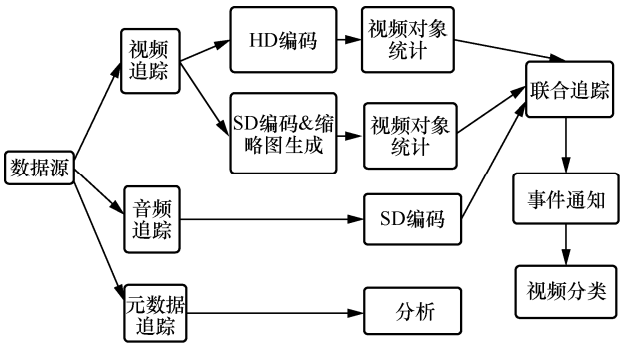


图1 视频处理应用程序的相关性任务

在目前的研究工作中，对相关性任务的调度研究工作不是很多，大致可以分为以下几个方面。

1) 任务在单服务器上的调度。给定一个应用程序图，文献[12]提出了一种启发式的程序分割方案，将相关性任务卸载到边缘服务器进行计算。文献[13]考虑了本地设备的能耗，并提出启发式算法，以在任务的截止期限和优先级约束下降低设备的能耗。以上工作是针对相关性任务在单MEC服务器的调度问题。

2) 任务在同构多服务器上的调度。目前，也存在多MEC服务器的工作研究。文献[14]忽略了MEC服务器之间的传输时间，首先定义了系统的总成本，包括任务的执行成本以及移动设备和边缘服务器之间的输入/输出数据的通信成本，研究在应用程序具有完成截止时间的约束下最小化系统成本的问题，并提出了通信增强的最新可能调度（CALPS, communication augmented latest possible scheduling）算法并获得近似解。文献[15]假设边缘节点的处理速度相同或边缘节点之间的传输时间相同，研究在车联网（IoV, Internet of vehicles）场景下的依赖性任务的卸载问题，通过设置优先级队列来确定调度任务的顺序，解决任务的相关性带来的挑战。文献[16]假设设备具有无限的处理速度，并且每个设备可以同时执行多个任务而不会影响每个任务的处理速度，并提出了一种完全多项式时间逼近算法以解决相关性任务卸载到多个设备的问题，目标是在资源成本约束的条件下最小化系统的总时延。但在现实生活中，移动设备的计算资源通常是有限的。文献[17]忽略了机器之间的传输时间，研究了一致相关机器上的工作流调度，并提出了基于速度的列表调度（SLS, speed-based list scheduling）算法。文献[18]忽略了处理器处理速度的差异性，提出了最早时间优先（ETF, earliest time first）算法来调度处理器之间具有通信时间的相关性任务。

3) 任务在异构多服务器上的调度。为了解决异构环境中相关性任务的调度问题，文献[11]提出了基于任务复制（TDB, task duplication-based）的算法，该算法的主要思想是在多台设备上复制任务，以便任务的结果可以在不同的设备上使用，用任务的执行时间换取数据的通信时间，目标是获得任务的最优卸载决策，使整个应用程序可以尽快完成。文献[19]提出了基于列表调度的异构最早完成时间（HEFT, heterogeneous earliest finish time）算法，该算法通过确定任务的优先级选择将任务卸载到最早完成时间的边缘节点上执行，但是，上述启发式

算法无法保证性能。目前，机器学习或深度强化学习已成为一个新的研究方向。文献[20]提出了一种新的任务调度算法，称为 QL-HEFT，它将 Q-learning 与 HEFT 算法相结合以减少任务的完成时间。具体来说，QL-HEFT 算法分为 2 个主要阶段：一是基于 Q-learning 的任务排序阶段，用于获得最优顺序；二是使用最早完成时间策略来获取任务的卸载方案阶段。文献[21]提出了一种基于任务重复的学习算法，称为 Lachesis，用于有向无环图（DAG, directed acyclic graph）任务的调度问题。Lachesis 算法首先使用专门设计的图卷积网络（GCN, graph convolutional network）感知作业之间的拓扑依赖关系，以选择最有可能执行的任务，然后将任务分配给特定的服务器，并考虑根据启发式方法复制其所有前驱任务。目前，深度强化学习和机器学习的方法已经成为解决 NP 难问题的一个很好的解决方案，但算法的效率在很大程度上取决于网络和样本数据。

目前终端设备的性能不断提高，而且有些任务必须在终端设备上才能完成。与上述文献研究不同，本文主要考虑了部分任务必须在终端设备上才能完成，研究多终端设备和多边缘节点的异构环境下如何在本地和边缘侧联合调度（JS, joint scheduling）相关性任务。考虑到终端设备上应用程序可能有完成截止时间，而终端设备的能耗有限，因此，将问题形式化为满足应用程序完成截止时间的约束下，最小化系统能耗。

本文主要的研究工作如下。

1) 研究相关性任务在本地和边缘节点的联合调度算法，并将问题形式化为在满足应用程序的完成截止时间约束的条件下最小化系统能耗。

2) 提出联合调度算法高效地求解该问题。仿真结果表明，本文算法具有很好的时间切换、功率分配和能效性能。与传统非 SWIPT 算法和非稳健算法进行对比，验证了本文算法的有效性。

3) 在随机生成 DAG 和分子动力学代码 DAG[22]上运行大量的仿真实验，将本文提出的 JS 算法与其他算法在应用程序的完成率和能耗两方面进行对比分析。

1　系统模型与问题建立

1.1　任务依赖模型

本文考虑系统中有一组终端设备 $\mathcal{N}=\{1,2,\cdots,N\}$，每个终端设备上可能存在多个相关任务，这些任务是不可分割的，每个任务只能卸载到一个边缘节点上执行，并且每个边缘节点在同一时刻只能执行一个任务，任务在执行过程中不能被中断。给定这些任务之间的优先级约束，可以构造多个 DAG 来表示所有任务之间的依赖关系。用 $G_{\mathcal{N}_i}(\mathcal{V},E)$ 表示终端设备 $\mathcal{N}_i$ 上应用程序的任务之间的依赖关系，其中，$\mathcal{V}=\{1,2,\cdots,V\}$ 表示终端设备上任务的集合，V 是 $\mathcal{N}_i$ 产生的所有任务的数量。每个任务 i 由一个二元组 $<B_i,D_i>$ 来表示，其中，B_i 表示执行任务的工作量，D_i 表示任务的数据量。此外，每个终端设备上的应用程序都有一个完成截止时间为 $T_{\mathcal{N}_i}^{\max}$。E 是一组边的集合，表示任务之间的优先级约束。DAG 中边的权重表示任务之间的数据传输量，用 $u_{i,j}$ 来表示。具体来说，当且仅当任务 i 是任务 j 的前驱时，任务 i 到任务 j 才会存在一条从 i 指向 j 的边。例如，$e(i,j)\in E$ 意味着任务 i 应该在任务 j 开始之前完成，并且任务 i 的输出数据传输到任务 j 所卸载到的边缘节点之后，任务 j 才可能开始执行。值得注意的是，一个任务可能存在多个前驱和多个后继任务。

1.2　网络模型

本文考虑一个包含多个边缘节点的 MEC 网络，使用 $\mathcal{K}=\{1,2,\cdots,K\}$ 来表示边缘节点的集合，其中 K 是边缘节点的数量。考虑到边缘节点的异构性，用 f_k 表示边缘节点 k 的处理速度。边缘节点之间通过各种网络（如主干网）相互连接，本文认为系统中所有的边缘节点之间都可以相互通信。边缘节点 k 与边缘节点 k' 之间传输每单位数据的传输时间记为 $\xi_{kk'}$，$\forall k,k'\in\mathcal{K}$。如果 $k=k'$，那么 $\xi_{kk'}=0$，因为相比于边缘节点之间的数据传输时间，边缘节点内部的传输时间可以忽略不计。

1.3　完成时间

由于任务可以在本地执行，也可以在边缘节点上执行，因此，任务的完成时间包含任务在本地执行的完成时间和任务在边缘节点上执行的完成时间 2 种计算模式。如果终端设备选择将任务 i 放在本地执行，那么任务 i 在本地的执行时间可以表示为

$$\mathrm{ET}_{i,\mathcal{N}_i}^{\mathrm{local}}=\frac{B_i}{f_{\mathcal{N}_i}},\quad \forall i\in\mathcal{V},\forall\mathcal{N}_i\in\mathcal{N} \tag{1}$$

其中，B_i 是执行任务 i 的工作量，$f_{\mathcal{N}_i}$ 是终端设备 $\mathcal{N}_i$ 执行任务时的实际处理速度。

由于终端设备的处理能力非常有限，在同一时间内，可以认为每个终端设备只能处理一个任务并且任务在执行过程中不能被中断。如果有多个任务需要在本地上执行，其他的任务需要在本地队列等候当前任务执行完之后才能开始。用 $\mathrm{RT}_{i,\mathcal{N}_i}^{\mathrm{local}}$ 表示任务 i 在终端设备 $\mathcal{N}_i$ 上的准备时间，即

$$\mathrm{RT}_{i,\mathcal{N}_i}^{\mathrm{local}} = \max_{j\in\mathrm{pred}(i)}\{(1-x_{j,k})\mathrm{FT}_{j,\mathcal{N}_i}^{\mathrm{local}} + x_{j,k}\mathrm{FT}_{j,k}^{\mathrm{edge}}\},\ \forall i\in\mathcal{V},\forall k\in\mathcal{K} \tag{2}$$

其中，$x_{j,k}$ 表示任务 j 是否在边缘节点 k 上执行，$x_{j,k}=1$ 表示任务 j 在边缘节点 k 上执行，$x_{j,k}=0$ 表示任务 j 不在边缘节点 k 上执行；$\mathrm{FT}_{j,\mathcal{N}_i}^{\mathrm{local}}$ 和 $\mathrm{FT}_{j,k}^{\mathrm{edge}}$ 分别表示任务 j 在本地的终端设备 $\mathcal{N}_i$ 和在边缘节点 k 上的完成时间。注意，由于本文只研究同一个应用程序上产生的任务，这些任务可能是相关的，而应用程序之间是相互独立的，而式(2)中的任务 j 是任务 i 的直接前驱，因此，任务 i 和任务 j 是由同一个应用程序产生的，为了方便表示，产生任务 j 的终端设备也用 $\mathcal{N}_i$ 来表示。因此，任务 i 在本地的开始时间可以表示为

$$\mathrm{ST}_{i,\mathcal{N}_i}^{\mathrm{local}} = \max\{\mathrm{RT}_{i,\mathcal{N}_i}^{\mathrm{local}}, \mathrm{AT}_{\mathcal{N}_i}\},\ \forall i\in\mathcal{V},\forall\mathcal{N}_i\in\mathcal{N} \tag{3}$$

其中，$\mathrm{AT}_{\mathcal{N}_i}$ 是终端设备 $\mathcal{N}_i$ 开始空闲的时间。那么，任务 i 在本地执行的完成时间 $\mathrm{FT}_{i,\mathcal{N}_i}^{\mathrm{local}}$ 可以表示为

$$\mathrm{FT}_{i,\mathcal{N}_i}^{\mathrm{local}} = \mathrm{ST}_{i,\mathcal{N}_i}^{\mathrm{local}} + \mathrm{ET}_{i,\mathcal{N}_i}^{\mathrm{local}},\ \forall i\in\mathcal{V},\forall\mathcal{N}_i\in\mathcal{N} \tag{4}$$

如果终端设备选择将任务 i 卸载到边缘节点 k 上执行，那么任务 i 在边缘节点 k 的执行时间可以表示为

$$\mathrm{ET}_{i,k}^{\mathrm{edge}} = \frac{B_i}{f_k},\forall i\in\mathcal{V},\forall k\in\mathcal{K} \tag{5}$$

同样地，本文认为每个边缘节点一次只能处理一个任务并且任务在执行过程中不能被中断。如果有多个任务卸载到同一个边缘节点上，除当前在边缘节点上执行的任务外，其他任务需要在边缘节点的队列中等候当前任务执行完之后才能开始。用 $\mathrm{RT}_{i,k}^{\mathrm{edge}}$ 表示任务 i 在边缘节点 k 上的准备时间，其可以表示为

$$\mathrm{RT}_{i,k}^{\mathrm{edge}} = \max_{j\in\mathrm{pred}(i)}\{(1-x_{j,k'})\mathrm{FT}_{j,\mathcal{N}_i}^{\mathrm{local}} + x_{j,k'}(\mathrm{FT}_{j,k'}^{\mathrm{edge}} + \xi_{k'k}u_{j,i})\},\ \forall i\in\mathcal{V},\forall k\in\mathcal{K},k'\in\mathcal{K} \tag{6}$$

其中，$\xi_{k'k}u_{j,i}$ 表示任务 j 和任务 i 分别在边缘节点 k 和 k' 上执行时的输出数据的传输时间。那么，任务 i 在边缘节点 k 的开始时间 $\mathrm{ST}_{i,k}^{\mathrm{edge}}$ 可以表示为

$$\mathrm{ST}_{i,k}^{\mathrm{edge}} = \max\{\mathrm{RT}_{i,k}^{\mathrm{edge}}, \mathrm{AT}_k\},\ \forall i\in\mathcal{V},k\in\mathcal{K} \tag{7}$$

其中，AT_k 是边缘节点 k 开始空闲的时间。因此，任务 i 在边缘节点 k 上的完成时间可以表示为

$$\mathrm{FT}_{i,k}^{\mathrm{edge}} = \mathrm{ST}_{i,k}^{\mathrm{edge}} + \mathrm{ET}_{i,k}^{\mathrm{edge}},\ \forall i\in\mathcal{V},k\in\mathcal{K} \tag{8}$$

1.4 系统能耗

系统能耗的计算包括计算能耗和传输能耗。任务 i 在本地执行的计算能耗可以表示为

$$\mathrm{EC}_{i,\mathcal{N}_i}^{\mathrm{local}} = \kappa B_i(f_{\mathcal{N}_i})^2,\ \forall i\in\mathcal{V},\forall\mathcal{N}_i\in\mathcal{N} \tag{9}$$

其中，κ 是能量系数，根据文献[23]，本章设置 $\kappa=10^{-28}$；$f_{\mathcal{N}_i}$ 的值一定要小于终端设备 $\mathcal{N}_i$ 的最大处理速度 $f_{\mathcal{N}_i}^{\max}$，通过调整 $f_{\mathcal{N}_i}$，可以获得不同的计算能耗。由于边缘节点通常由无线访问接入点（AP, access point）或基站供电，因此本文不考虑在边缘节点上执行任务的能耗。

网络通信采用正交频分多址（OFDMA, orthogonal frequency division multiple access）技术，假设分配给所有终端设备的频谱是正交的，那么终端设备 $\mathcal{N}_i$ 将任务 i 传输到边缘节点 k 的传输速率可以表示为

$$R_{\mathcal{N}_i,k} = W_{\mathcal{N}_i,k}\log\left(1+\frac{P_{\mathcal{N}_i}l_{\mathcal{N}_i,k}^{-\beta}}{\sigma^2}\right),\forall k\in\mathcal{K},\forall\mathcal{N}_i\in\mathcal{N} \tag{10}$$

如果任务 i 卸载到边缘节点上执行，传输能耗仅计算上传任务的能耗，将任务卸载到边缘节点上执行的传输时间可以表示为

$$T_{i,k} = \frac{D_i}{R_{\mathcal{N}_i,k}},\forall i\in\mathcal{V},\forall k\in\mathcal{K},\forall\mathcal{N}_i\in\mathcal{N} \tag{11}$$

因此，任务 i 卸载到边缘节点 k 时的传输能耗可以表示为

$$\mathrm{EC}_{\mathcal{N}_i,k}^{\mathrm{tran}} = T_{i,k}P_{\mathcal{N}_i},\ \forall i\in\mathcal{V},k\in\mathcal{K},\forall\mathcal{N}_i\in\mathcal{N} \tag{12}$$

其中，$T_{i,k}$ 是任务上传到边缘节点 k 的传输时间，$P_{\mathcal{N}_i}$ 是终端设备 $\mathcal{N}_i$ 的传输功率。这里不考虑任务传回终端设备的能耗，因为任务的数据结果是非常小的，将数据结果传回终端设备的下行链路传输时间也是非常短的，所以此部分能耗可以忽略不计。

经过上述的讨论，可以得出完成系统中所有任务所消耗的能量可以表示为

$$\mathrm{EC}=\sum_{\mathcal{N}_i\in\mathcal{N}}\sum_{i\in\mathcal{N}_i}(1-x_{i,k})\mathrm{EC}^{\mathrm{local}}_{i,\mathcal{N}_i}+\sum_{\mathcal{N}_i\in\mathcal{N}}\sum_{i\in\mathcal{N}_i}x_{i,k}\mathrm{EC}^{\mathrm{tran}}_{\mathcal{N}_i,k} \quad (13)$$

系统中每个应用程序产生的所有任务的完成时间可以表示为

$$\mathrm{FT}_{\mathcal{N}_i}=\max_{i\in V}\left((1-x_{i,k})\mathrm{FT}^{\mathrm{local}}_{i,\mathcal{N}_i}+x_{i,k}\mathrm{FT}^{\mathrm{edge}}_{i,k}\right),\forall\mathcal{N}_i\in\mathcal{N} \quad (14)$$

当终端设备以更高的 CPU 频率执行任务时，系统的性能会随着应用程序的完成时间的减少而提高，但代价是消耗更多的能源。能耗和任务的完成时间这 2 个目标在本质上是相互冲突的，因此，本文面临能耗和任务完成时间的平衡问题。由于在现实生活中，应用程序的完成时间可能会有一个完成截止时间，因此本节将系统能耗作为研究目标，应用程序的完成截止时间作为约束条件，在尽量满足每个应用程序的完成截止时间的条件下，最小化系统能耗，如果即使以最大的执行频率执行任务，应用程序也无法在完成截止时间前完成，那么该应用程序的任务将会被丢弃。最终，本节将问题形式化为

$$\begin{aligned}&\min_{S_1^{U(K+N)}}\ \mathrm{EC}\\&\text{s.t.}\mathrm{FT}_{\mathcal{N}_i}\leqslant T^{\max}_{\mathcal{N}_i},\forall\mathcal{N}_i\in\mathcal{N}\end{aligned} \quad (15)$$

其中，$T^{\max}_{\mathcal{N}_i}$ 是应用程序 $\mathcal{N}_i$ 的完成截止时间。

2　问题求解与算法设计

本节提出了相关性任务在本地和边缘节点上的 JS 算法。JS 算法的输入是终端设备的集合 $\mathcal{N}$ 和系统中的所有 DAG，输出是在满足应用程序的完成截止时间的约束条件下达到最小化系统能耗的任务调度策略 $S_1^{U(K+N)}$。JS 算法的描述如算法 1 所示。

算法 1　JS 算法

输入　用户终端设备的集合 $\mathcal{N}$，DAG

输出　调度策略 $S_1^{U(K+N)}$

1) $S^{U(K+N)}\leftarrow\varnothing$；　$S_1^{U(K+N)}\leftarrow\varnothing$；

2) $S^{U(K+N)}\leftarrow\mathrm{TPS}(\mathcal{N},\ \mathrm{DAG})$；

3) $S_1^{U(K+N)}\leftarrow\mathrm{SFSEC}(S^{U(K+N)})$；

4)return $S_1^{U(K+N)}$；

接下来，详细介绍 TPS 算法。该算法在没有考虑系统能耗的情况下，终端设备和边缘节点均以最大的处理速度执行任务，得到调度策略 $S^{U(K+N)}$。TPS 算法的详细步骤介绍如下。

1) 确定应用程序的优先级阶段

为了确保所有应用程序在各自的完成截止时间前完成，首先构建一个应用程序的优先级队列。

2) 确定任务的优先级阶段

由于属于同一应用程序的任务之间可能存在相关性，因此，需要确定单个任务的完成时间约束。每个任务的完成时间约束可以通过计算任务的最迟完成时间来获得。这里将单个任务的最迟完成时间看作是其单独的完成时间的约束。为此，构建了一个任务的优先级队列 TQ，任务的优先级队列以任务的优先级递减的顺序保持下去。任务的优先级的计算方式如下。

首先，任务 i 的最迟完成时间是任务 i 在本地或边缘节点上完成执行的最迟时间。用 LFT_i 表示任务 i 的最迟完成时间，即

$$\mathrm{LFT}_i=\min_{j\in\mathrm{succ}(i)}(\mathrm{LFT}_j-\mathrm{ET}^{\min}_j),\forall i\in\mathcal{V} \quad (16)$$

其中，任务 j 是任务 i 的后继，$\mathrm{ET}^{\min}_j$ 是完成任务 j 所需的最少执行时间，可以表示为

$$\mathrm{ET}^{\min}_j=\min_{k\in\mathcal{K},\mathcal{N}_j\in\mathcal{N}}(\mathrm{ET}^{\mathrm{local}}_{j,\mathcal{N}_i},\mathrm{ET}^{\mathrm{edge}}_{j,k}),\forall j\in\mathcal{V} \quad (17)$$

对于产生任务 i 的应用程序 $\mathcal{N}_i$ 来说，它的完成截止时间是 $T^{\max}_{\mathcal{N}_i}$。因此，对于此应用程序上的最后一个任务的完成截止时间也是 $T^{\max}_{\mathcal{N}_i}$。如果应用程序中的任意一个任务无法在其最迟完成时间完成，那么此应用程序是不可能在完成截止时间 $T^{\max}_{\mathcal{N}_i}$ 内完成的。

任务 i 的最迟开始时间是任务 i 在本地或边缘节点上开始执行的最迟时间。用 LST_i 表示任务 i 的最迟开始时间，即

$$\mathrm{LST}_i=\mathrm{LFT}_i-\mathrm{ET}^{\min}_i,\forall i\in\mathcal{V} \quad (18)$$

另外，越紧急的任务应该有越高的优先级，因此，最迟开始时间被用作优先级的度量。也就是说，对于所有的任务，具有越小的最迟开始时间的任务的优先级越高。

如果任务 i 刚好在最迟完成时间 LFT_i 完成，那么任务 i 之后的任务的调度时间是非常紧张的。因此，定义 LFT'_i 为任务 i 的松弛最迟完成时间，其可以表示为

$$\mathrm{LFT}'_i=\min_{j\in\mathrm{succ}(i)}(\mathrm{LFT}_j-\mathrm{ET}^{\max}_j),\forall i\in\mathcal{V} \quad (19)$$

$$\mathrm{ET}_j^{\max}=\max_{k\in\mathcal{K},\mathcal{N}_j\in\mathcal{N}}(\mathrm{ET}_{j,\mathcal{N}_i}^{\mathrm{local}},\mathrm{ET}_{j,k}^{\mathrm{edge}}),\forall j\in\mathcal{V} \quad (20)$$

由于任务 i 的松弛最迟完成时间 LFT_i' 通常早于其最迟完成时间 LFT_i，如果所有任务都在其松弛最迟完成时间之前完成，那么任务可以总是在其最迟完成时间之前完成。这就可以确保应用程序可以在其完成截止时间之前完成。因此，对于任务 i，本节用 LFT_i' 代替 LFT_i。

3) 任务调度的阶段

首先选择优先级最高的任务，如果任务必须在本地执行，那么将其放在本地计算；如果任务没有要求在本地执行，则选择在本地或边缘节点上执行使其完成时间最小。如果只有一个应用程序，则可以根据任务的优先级队列中的顺序来调度任务；当有多个应用程序时，如果具有较低优先级的任务比其他具有较高优先级的任务更早准备好执行（即有可能较低优先级的任务的所有直接前驱都已被调度并将输出数据传输至所需的任务），则更愿意先调度已经准备好的任务，而不是按照优先级队列的顺序进行调度。因此，需要定义一个任务辅助队列 AQ 来存储优先级高于当前任务但目前还没有准备好调度的任务。TPS 算法的详细描述如算法 2 所示。

算法 2 TPS 算法

输入 终端设备的集合 $\mathcal{N}$，任务图 DAG

输出 任务的调度策略 $S^{U(K+N)}$

1) $S^{U(K+N)}\leftarrow\varnothing$，TQ $\leftarrow\varnothing$，AQ $\leftarrow\varnothing$；
2) 根据步骤 1)得到应用程序的优先级；
3) 根据步骤 2)计算 LST_i 得到所有任务的优先级队列 TQ；
4) while TQ $\neq\varnothing$ do
5) $i\leftarrow$ TQ.head；
6) 根据式(2)和式(7)分别计算 $\mathrm{RT}_{i,\mathcal{N}_i}^{\mathrm{local}}$ 和 $\mathrm{RT}_{i,k}^{\mathrm{edge}}$；
7) if 任务 i 没有准备好调度 then
8) AQ = AQ$\cup\{i\}$；
9) else
10) call Schedule Task (i)；
11) end if
12) update i；
13) end while
14) function Schedule Task (i)
15) 调用 Get Finish Time (i)；
16) if $\exists k,\mathrm{s.t.}\mathrm{FT}_{i,k}^{\mathrm{edge}}<\mathrm{FT}_{i,\mathcal{N}_i}^{\mathrm{local}}$ then
17) $k\leftarrow\min_{k\in\mathcal{K}}\mathrm{FT}_{i,k}^{\mathrm{edge}}$；
18) 将任务 i 卸载到边缘节点 k 上；
19) else
20) 将任务 i 放在本地 $\mathcal{N}_i$ 上执行；
21) end if
22) if AQ = ∅ or 任务 i 位于队列 TQ 的头部 then
23) 调用 Schedule (i)；
24) end if
25) for all $j\notin\mathcal{N}_i$ in AQ do
26) 调用 Get Finish Time (j)；
27) if $\exists k,\mathrm{s.t.}\,\mathrm{FT}_{j,k}^{\mathrm{edge}}>\mathrm{LFT}_j'$ and $\mathrm{FT}_{j,\mathcal{N}_j}^{\mathrm{local}}>\mathrm{LFT}_j'$ then
28) 调用 Schedule (j)；
29) end if
30) end for
31) 调用 Schedule (i)；
32) end function
33) function Get Finish Time (i)
34) if 任务 i 必须在本地执行 then
35) 根据式(4)计算 $\mathrm{FT}_{i,\mathcal{N}_i}^{\mathrm{local}}$；
36) else
37) for $k=1$ to K do
38) 根据式(9)计算 $\mathrm{FT}_{i,k}^{\mathrm{edge}}$；
39) end for
40) 根据式(4)计算 $\mathrm{FT}_{i,\mathcal{N}_i}^{\mathrm{local}}$；
41) end if
42) return $\mathrm{FT}_{i,k}^{\mathrm{edge}}$，$\mathrm{FT}_{i,\mathcal{N}_i}^{\mathrm{local}}$；
43) end function
44) function Schedule (i)
45) 选择将任务 i 调度到边缘节点 k 或在本地设备 $\mathcal{N}_i$ 上执行；
46) 更新 $S^{U(K+N)}$；
47) TQ $\leftarrow$ TQ $\setminus\{i\}$；
48) end function

TPS 算法的输入是终端设备的集合 $\mathcal{N}$ 以及任务图 DAG，输出是任务的调度策略 $S^{U(K+N)}$，其中，U 是所有终端设备上的应用程序产生的任务的数量。首先，对调度策略 $S^{U(K+N)}$、任务的辅助队列 AQ 和任务的优先级队列 TQ 进行初始化。然后对应用

程序和任务的优先级进行排序，当队列 TQ 不为空时，将队列头部的任务元素赋为任务 i，分别计算任务 i 在本地和所有的边缘节点上的准备时间，如果任务 i 没有准备好，就将其放入辅助队列 AQ 中；否则，调用函数 Schedule Task (i)。在函数 Schedule Task (i)中，调用函数 Get Finish Time (i)得到任务 i 分别在本地和边缘节点上的完成时间，如果任务 i 在边缘节点上的完成时间小于在本地的完成时间，就将任务 i 卸载到具有最小完成时间的边缘节点 k 上执行，否则，任务 i 将在本地执行。如果辅助队列 AQ 为空或任务 i 是队列 AQ 的头部元素，也就是当前任务 i 的优先级最高，就调用函数 Schedule (i)；否则，计算所有与产生任务 i 不同的用户终端设备上的应用程序的任务 j 在本地或边缘节点上的完成时间。如果任务 j 的完成时间超过了其松弛完成时间，就对任务 j 调用函数 Schedule (j)，再对任务 i 调用函数 Schedule (i)。在函数 Schedule (i)中，将任务 i 调度到本地 $\mathcal{N}_i$ 或边缘节点上 k 上执行，并更新调度策略 $S^{U(K+N)}$ 和任务优先级队列 TQ，最后将任务 i 在队列中移除。

为了进一步降低系统能耗，接下来采用逐步频率缩放的方法，但降低终端设备的执行频率可能会导致整个系统中任务的完成时间增加。对于现实世界的应用程序，必须满足应用程序的完成截止时间的约束条件。因此，本文在算法 2 得到的调度策略 $S^{U(K+N)}$ 的基础上提出了 SFSEC 算法，通过调整终端设备的执行频率，达到最小化系统能耗的目的。

SFSEC 算法的思想在于不改变任务是在本地执行还是卸载到边缘节点上执行的决策，从而避免对任务的重新决策过程。由于任务之间的优先级约束和数据的传输时延，任务在执行之前可能会在本地或边缘节点上产生空闲时间。对于在边缘节点 k 或终端设备上 $\mathcal{N}_i$ 调度任务 i，可以使用以下的方式表示任务 i 的空闲时间 SlackTime

$$\text{SlackTime}_i = \min(\text{ST}_m, \min_{j\in\text{succ}(i)} \text{ST}_j) \tag{21}$$

其中，m 表示与任务 i 卸载到同一个边缘节点或用户终端设备上即将调度的任务，ST_m 表示任务 m 的开始时间。也就是说，SFSEC 算法是在不影响即将调度的任务 m 的开始时间和任务 i 所有的后继任务 j 的开始时间的情况下，最大可能地延迟任务 i 的完成时间。此时，如果任务 i 是 DAG 中的退出任务，那么，任务 i 的空闲时间 SlackTime 可以表示为

$$\text{SlackTime}_i = T_{\mathcal{N}_i}^{\max} - \text{AFT}_i \tag{22}$$

其中，AFT_i 是任务 i 的实际完成时间，并且它是任务 i 在调度执行后得到的。

SFSEC 算法的详细描述如算法 3 所示。

算法 3　SFSEC 算法

输入　算法 2 得到的调度策略 $S^{U(K+N)}$

输出　调整执行频率后得到的调度策略 $S_1^{U(K+N)}$

1) $S_1^{U(K+N)} \leftarrow S^{U(K+N)}$;
2) for all $i \in \mathcal{V}$ do
3) if $x_{i,k} = 0, \forall k \in \mathcal{K}$ then
4) $f_{\mathcal{N}_i} \leftarrow \left\lceil \frac{f_{\mathcal{N}_i}^{\max}}{2} \right\rceil$;
5) while $f_{\mathcal{N}_i} > 0$ do
6) 根据式(4)计算 $\text{FT}_{i,\mathcal{N}_i}^{\text{local}}$;
7) if $\exists m \in \mathcal{N}_i$ then
8) $T_1 \leftarrow \text{ST}_m$;
9) else
10) if 任务 i 是终端设备上的最后一个任务 then
11) $T_2 \leftarrow T_{\mathcal{N}_i}^{\max}$;
12) else
13) $T_2 \leftarrow \min_{j\in\text{succ}(i)} \text{ST}_j$;
14) end if
15) if $\text{FT}_{i,\mathcal{N}_i}^{\text{local}} \leqslant T_1 \,\&\, \text{FT}_{i,\mathcal{N}_i}^{\text{local}} \leqslant T_2$ then
16) $f_{\mathcal{N}_i} \leftarrow \left\lceil \frac{f_{\mathcal{N}_i}}{2} \right\rceil$;
17) continue;
18) else
19) $x_{i,k}^{'} \leftarrow f_{\mathcal{N}_i}$;
20) $\text{AFT}_i \leftarrow \text{FT}_{i,\mathcal{N}_i}^{\text{local}}$;
21) end if
22) end if
23) update $S_1^{U(K+N)}$;
24) end while
25) end if
26) end for
27) return $S_1^{U(K+N)}$;

SFSEC 算法的输入是算法 2 得到的调度序列 $S^{U(K+N)}$，输出是在不改变任务的卸载决策的情况下，对在终端设备上执行的任务调整执行频率后得到调度策略 $S_1^{U(K+N)}$。首先，遍历所有在终端设备上处理的任务 i，以二分搜索频率的方式迭代计算任务 i 的完成时间。如果存在任务 m，m 是与任务 i 在同一个终端设备上的即将调度的任务，将任务 m 的开始时间赋为 T_1；否则，判断任务 i 是否是 DAG 的退出任务，如果是，将 $T_{N_i}^{\max}$ 赋为 T_2，如果不是，将任务 i 所有的后继任务 j 的最小开始时间赋为 T_2。如果以频率 f_{N_i} 执行的任务 i 的完成时间 $\mathrm{FT}_{i,N_i}^{\mathrm{local}}$ 不超过 T_1 和 T_2，则继续调频；否则，认为以频率 f_{N_i} 执行任务 i 是可以满足不影响其他任务的开始时间并保证应用程序能在完成截止时间前完成的最小执行频率，之后更新任务 i 的实际完成时间 AFT_i，最后更新调度策略 $S_1^{U(K+N)}$，并返回 $S_1^{U(K+N)}$。

算法 2 的时间复杂度的具体分析如下。确定应用程序优先级队列阶段的时间复杂度由第 2)行确定，对应用程序排序的时间复杂度为 $O(N\log N)$；确定任务优先级队列阶段的时间复杂度由第 3)行决定，其计算复杂度为 $O(U)$，因此对任务排序过程的时间复杂度为 $O(U\log U)$。在任务调度阶段，主要的操作是第 4)~13)行的 while 循环，while 循环的时间复杂度主要由第 4)行和第 10)行决定，第 4)行的时间复杂度为 $O(U)$，第 10)行的时间复杂度由第 14)~32)行决定，其时间复杂度为 $O(UK)$，则任务调度阶段的时间复杂度为 $O(U^2K)$。因此，算法 2 的时间复杂度为 $O(N\log N)+O(U\log U)+O(U^2K)=O(U^2K)$。

算法 3 的时间复杂度的具体分析如下：逐步缩放频率阶段的时间复杂度为 $O\left(U\log^{f_{N_i}^{\max}}\right)$。因此，算法 3 的时间复杂度为 $O(U^2K)+O(U\log^{f_{N_i}^{\max}})=O(U^2K)$。

3 仿真实验

3.1 仿真实验设置

实验参数设置如表 1 所示，其中 V 是所有的终端设备产生的任务数量。每个应用程序的完成截止时间在[0.5,1.5] s 随机产生，假设每个终端设备的执行频率在[1,10] MHz 随机生成，最大执行频率在[10,20] MHz 随机生成，每个应用程序上产生的任务至少有 $\frac{1}{3}$ 需要在本地执行。

表 1 参数设置

参数	随机生成 DAG	分子动力学代码 DAG
B_i /GB	[0.1, 0.2]	[0.1, 0.2]
V /个	450	41
$\lvert\mathcal{N}\rvert$	10	3
D_i /KB	[100, 300]	[100, 300]
$u_{i,j}$ /KB	[20, 60]	[20, 60]
$\xi_{kk'}$ /(ms·KB^{-1})	[2, 5]	[2, 5]
K	40	4
f_k /GHz	[0.1, 1]	[0.1, 1]
$W_{N_i,k}$ /MHz	5	5
P_{N_i} /mW	100	100
$l_{N_i,k}$ /m	[3, 15]	[3, 15]
β /dB	11	11
σ^2 /mW	10^{-10}	10^{-10}
直接后继的数量	[1, 4]	—

本节主要关注应用程序的完成率和系统能耗的比较。其中，对于分子动力学代码 DAG，其 DAG 任务的数量是固定的，结构也是固定的，比较适合小规模的 DAG 任务。而对于随机生成 DAG，其 DAG 任务的数量和结构都是随机的，适合大规模的 DAG 任务。本节主要将 JS 算法与以下算法进行比较。

1) 随机算法（Random）。将必须在本地执行的任务放在本地终端设备上执行，其他任务随机选择在本地或边缘节点上执行。

2) 贪心算法（Greedy）。每次选择位于队列头部的任务，确定该任务是否必须在本地完成，如果不是，将此任务卸载到最早完成时间的边缘节点上执行。

3) TDB 算法[11]。该算法的思路是将所有任务划分为不同的任务集群，然后将同一个任务集群卸载到最早完成时间的边缘节点上执行，最后在多个边缘节点上复制任务的前驱，使前驱的数据结果可以在多个边缘节点上使用，即以执行时间换传输时间。

4) HEFT 算法[19]。该算法包括 2 个步骤：首先是确定任务的优先级阶段，计算每个任务的平均执行时间和平均传输时间之和，值越大，任务优先级越高；然后是边缘节点的选择阶段，选择目前优先级最高且没有前驱的任务，然后将选中的任务卸载到最早完成时间的边缘节点上执行。

HEFT、TDB 算法与 JS 算法只在完成率方面进行了对比，Random、Greedy 算法与 JS 算法在完成率和能耗方面都进行了对比。

3.2　应用程序完成率的仿真结果

第一组实验模拟通过改变任务的数量来分析应用程序的完成率，实验结果如图 2 所示。随着任务的数量的增加，所有算法下的应用程序的完成率都随之降低，这是因为设备要处理的任务工作量更多。值得注意的是，当任务的数量较少时，JS 算法、Greedy 算法、HEFT 算法和 TDB 算法下的应用程序的完成率都接近 1。例如，当任务的数量为 50 个时，JS 算法、Greedy 算法、HEFT 算法和 TDB 算法下所有的应用程序都能在其完成截止时间前完成，而 Random 算法下，只有约 35%的应用程序能在其完成截止时间前完成。随着任务的数量的增加，Greedy 算法和 JS 算法的完成率的差距越来越大，这是因为 Greedy 算法并未考虑到应用程序的完成截止时间的约束条件，每次贪婪地选择将任务卸载到最早完成时间的边缘节点上执行。而 TDB 算法和 JS 算法完成率的差距也逐步显现，这是因为 TDB 算法可能会复制任务的前驱并在不同的边缘节点上多次执行。此外，HEFT 算法在任务排序阶段考虑边缘节点处理速度的异构性，本质上，HEFT 算法专注于将任务卸载到最早完成时间的边缘节点上，当任务数量较少时，HEFT 算法可以达到和 JS 算法接近的性能，但是随着任务的数量的增加，性能差距越来越大，因为无论是卸载到边缘节点的任务的工作量增加还是任务的相关性都会导致任务的等待时间增加。

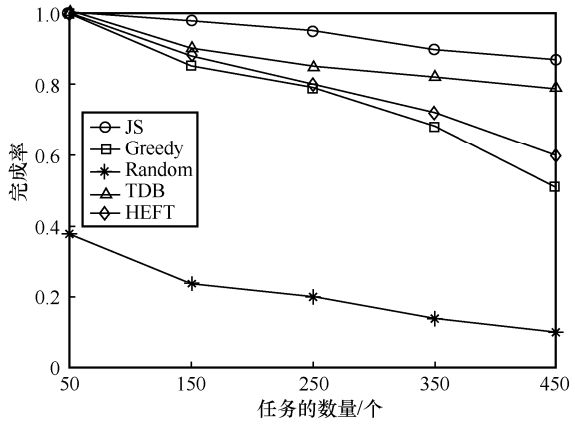


图 2　完成率与任务的数量的关系

第二组实验分别模拟在随机生成 DAG 结构（图 3(a)）和分子动力学代码 DAG 结构（图 3(b)）上通过改变边缘节点的数量来分析应用程序的完成率。随着边缘节点的数量的增加，所有算法下的应用程序的完成率都随之增加。显然，随着边缘节点的数量的增加，可以选择将任务卸载到空闲的边缘节点上执行，任务的完成时间会相应地减少，从而会有更多的应用程序能在其完成截止时间前完成。值得注意的是，当边缘节点的数量为 35 个时，Random 算法在随机生成 DAG 结构下的应用程序的完成率几乎为 0。虽然 JS 和 Greedy 算法的完成率也不是很高，但 JS 算法在 2 种不同的 DAG 结构下的完成率达到了 68%和 81%。当边缘节点的数量达到 55 个及以上时，JS 算法的完成率达到了 100%，这是因为 JS 算法考虑了应用程序的完成截止时间，当系统中有足够的边缘节点可以执行任务时，能考虑到每个应用程序的完成截止时间约束。当边缘节点的数量较少，比如为 2 时，HEFT 算法可以达到最佳性能，完成率和 JS 算法差距很小。HEFT 算法选择将任务卸载到最早完成时间的边缘节点上执行，即使处理速度慢，也可以分担待执行任务的工作量。而 TDB 和 Greedy 算法实现的性能差别不是很大，因为大部分任务都会被卸载到同一个边缘节点上执行，减少了前驱执行的次数。

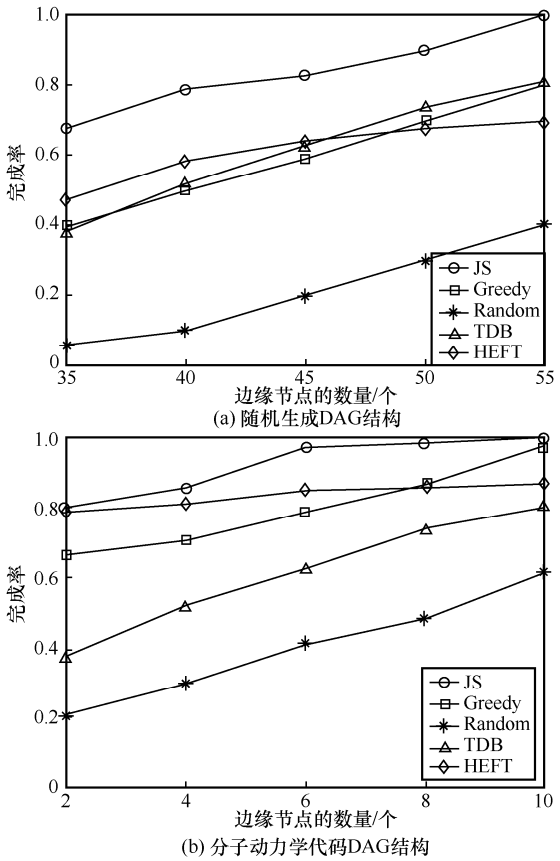


图 3　完成率与边缘节点的数量的关系

第三组实验模拟通过改变应用程序的完成截止时间来比较应用程序的完成率，实验结果如图 4 所示。其中，横坐标表示应用程序的完成截止时间与初始的应用程序的完成截止时间的比值。随着完成截止时间的延迟，所有算法下的应用程序的完成率随之增加。显然，由于应用程序的完成截止时间增加，原先不能在完成截止时间内完成的应用程序有希望在当前的完成截止时间内完成，因此应用程序的完成率会增加。当应用程序的完成截止时间为其初始值的一半时，Random 算法下的完成率均为 0，这是因为所有应用程序的完成时间更加紧张。在随机生成 DAG 和分子动力学代码 DAG 结构下，当完成截止时间增加到初始值的 1.50 倍和 1.25 倍时，JS 算法下的所有应用程序均能在其完成截止时间内完成。而当应用程序的完成截止时间足够宽裕时，Greedy 算法最终也能达到百分百的完成率。在分子动力学代码 DAG 结构下，TDB 算法效果虽然比 Greedy 算法好，但依然不理想，这是因为虽然其降低了执行时间，但任务仍然会被多次执行。而 HEFT 算法随着应用程序的完成截止时间的增加，效果却变得比 Greedy 算法差，这是因为 HEFT 有排序过程。

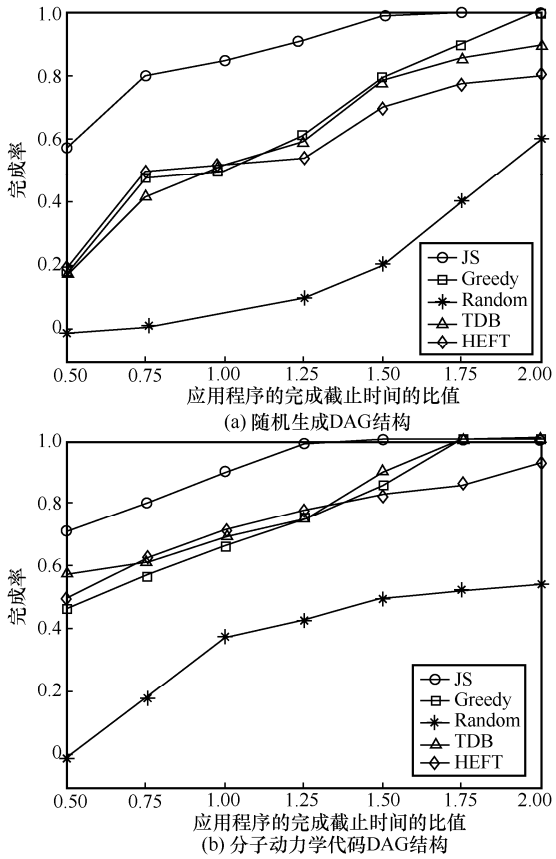


图 4 完成率与应用程序的完成截止时间的比值的关系

3.3 系统能耗的仿真结果

第一组实验模拟通过改变任务的数量来分析系统能耗，实验结果如图 5 所示。随着任务的数量的增加，所有算法下的系统能耗都随之增加。显然，任务的数量的增加使本地或边缘节点上执行任务的数据量增加，从而导致系统能耗的增加。由于 JS 算法使用了 SFSEC 算法来调整执行频率，因此 JS 算法始终能实现较低的系统能耗，而 Random 和 Greedy 算法不考虑系统能耗，随着任务的数量的增加，系统能耗几乎呈线性规律增长，两者在系统能耗的差距很大程度上是由任务的传输能耗造成的。例如，当任务的数量为 450 个时，相比于 Random 和 Greedy 算法，JS 算法分别降低了 50%和 43%的系统能耗。

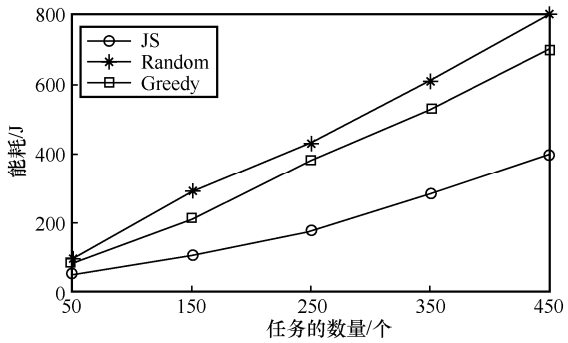


图 5 系统能耗与任务的数量的关系

第二组实验分别模拟在随机生成 DAG 结构和分子动力学代码 DAG 结构上通过改变边缘节点的数量来分析系统能耗，实验结果如图 6 所示。随着边缘节点的数量的增加，JS 算法实现的系统能耗随之减少，而其他 2 种算法实现的系统能耗变化不大，这是因为随着边缘节点的数量的增加，任务可以卸载到更多的边缘节点上执行，任务的执行时间会减少，设备可以通过降低执行频率使系统能耗减少。例如，当边缘节点的数量为 50 时，相比于 Random 和 Greedy 算法，JS 算法分别降低了 60%和 54%的系统能耗。

第三组实验分别模拟在随机生成 DAG 结构和分子动力学代码 DAG 结构上通过改变应用程序的完成截止时间的比值来分析系统能耗，实验结果如图 7 所示。随着完成截止时间的延迟，JS 算法实现的系统能耗随之降低，这是因为推迟了应用程序的完成截止时间，SFSEC 算法在调整执行频率时可以进一步降低频率，从而降低系统能耗。但 Random 和 Greedy 算法实现的系统能耗几乎没有变化，这是因为在这 2 种算法中不考虑系统能耗，每次调度

都选择在本地或者边缘节点上以最大的频率执行。例如，当应用程序的完成截止时间为初始值的 1.5 倍时，相比于 Random 和 Greedy 算法，JS 算法大约降低了 68%的系统能耗。

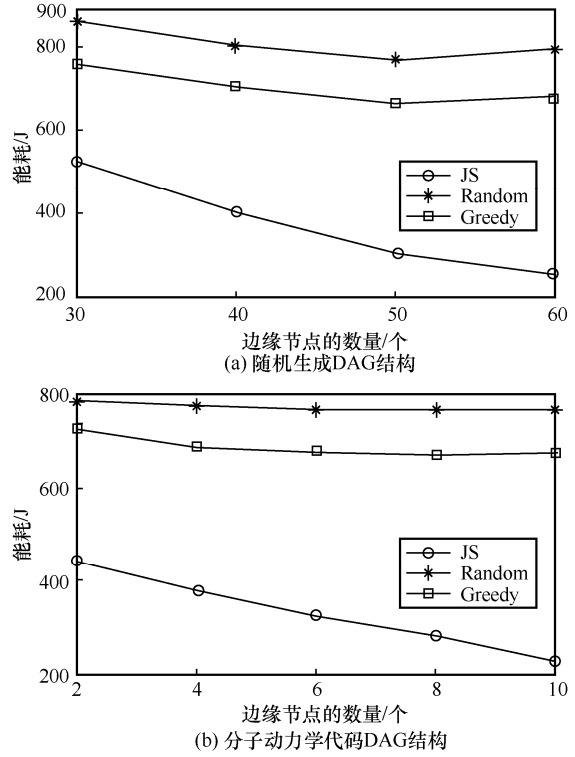


图 6　系统能耗与边缘节点的数量的关系

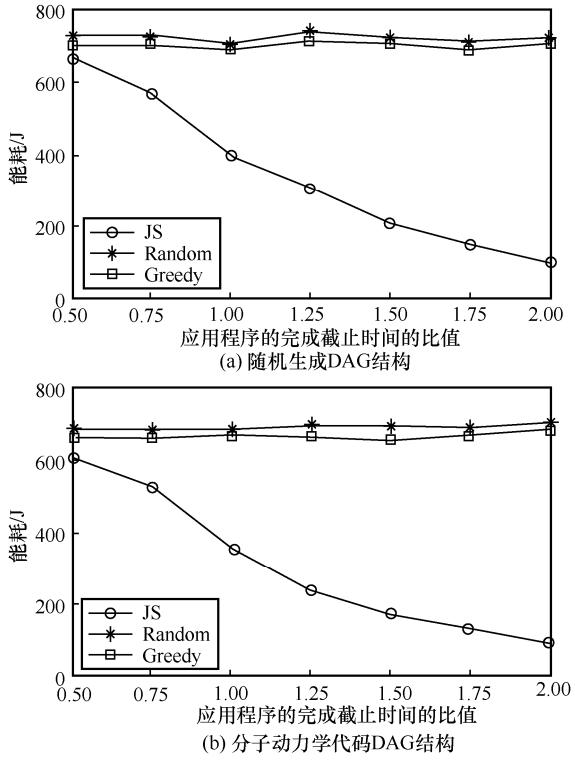


图 7　系统能耗与应用程序的完成截止时间的关系

4　结束语

本文研究了相关性任务在本地和边缘侧的联合调度方法，并最小化了系统能耗，提出了 JS 算法，最后通过仿真实验分析了 JS 算法在应用程序的完成率和系统能耗两方面的性能。仿真结果表明，JS 算法在应用程序的完成率上优于 Random、Greedy、HEFT 和 TDB 算法，并且与 Random、Greedy 算法相比较至少可以节省 43%的系统能耗。

参考文献：

[1] MACH P, BECVAR Z. Mobile edge computing: a survey on architecture and computation offloading[J]. IEEE Communications Surveys & Tutorials, 2017, 19(3): 1628-1656.

[2] XU J, CHEN L X, ZHOU P. Joint service caching and task offloading for mobile edge computing in dense networks[C]//Proceedings of IEEE INFOCOM 2018 - IEEE Conference on Computer Communications. Piscataway: IEEE Press, 2018: 207-215.

[3] SHI W S, CAO J, ZHANG Q, et al. Edge computing: vision and challenges[J]. IEEE Internet of Things Journal, 2016, 3(5): 637-646.

[4] CHEN M, HAO Y X. Task offloading for mobile edge computing in software defined ultra-dense network[J]. IEEE Journal on Selected Areas in Communications, 2018, 36(3): 587-597.

[5] ABBAS N, ZHANG Y, TAHERKORDI A, et al. Mobile edge computing: a survey[J]. IEEE Internet of Things Journal, 2018, 5(1): 450-465.

[6] SATYANARAYANAN M, BAHL P, CACERES R, et al. The case for VM-based cloudlets in mobile computing[J]. IEEE Pervasive Computing, 2009, 8(4): 14-23.

[7] LI Y Q, WANG X, GAN X Y, et al. Learning-aided computation offloading for trusted collaborative mobile edge computing[J]. IEEE Transactions on Mobile Computing, 2020, 19(12): 2833-2849.

[8] ESHRAGHI N, LIANG B. Joint offloading decision and resource allocation with uncertain task computing requirement[C]//Proceedings of IEEE INFOCOM 2019 - IEEE Conference on Computer Communications. Piscataway: IEEE Press, 2019: 1414-1422.

[9] TONG L, LI Y, GAO W. A hierarchical edge cloud architecture for mobile computing[C]//Proceedings of IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications. Piscataway: IEEE Press, 2016: 1-9.

[10] ZHAO G M, XU H L, ZHAO Y M, et al. Offloading dependent tasks in mobile edge computing with service caching[C]//Proceedings of IEEE INFOCOM 2020 - IEEE Conference on Computer Communications. Piscataway: IEEE Press, 2020: 1997-2006.

[11] HE K, MENG X Z, PAN Z Z, et al. A novel task-duplication based clustering algorithm for heterogeneous computing environments[J]. IEEE Transactions on Parallel and Distributed Systems, 2019, 30(1): 2-14.

[12] JIA M K, CAO J N, YANG L. Heuristic offloading of concurrent tasks

for computation-intensive applications in mobile cloud computing[C]//Proceedings of 2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). Piscataway: IEEE Press, 2014: 352-357.

[13] GUO S T, XIAO B, YANG Y Y, et al. Energy-efficient dynamic offloading and resource scheduling in mobile cloud computing[C]//Proceedings of IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications. Piscataway: IEEE Press, 2016: 1-9.

[14] SUNDAR S, LIANG B. Communication augmented latest possible scheduling for cloud computing with delay constraint and task dependency[C]//Proceedings of 2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). Piscataway: IEEE Press, 2016: 1009-1014.

[15] LIU Y J, WANG S G, ZHAO Q L, et al. Dependency-aware task scheduling in vehicular edge computing[J]. IEEE Internet of Things Journal, 2020, 7(6): 4961-4971.

[16] KAO Y H, KRISHNAMACHARI B, RA M R, et al. Hermes: Latency optimal task assignment for resource-constrained mobile computing[C]//Proceedings of 2015 IEEE Conference on Computer Communications (INFOCOM). Piscataway: IEEE Press, 2015: 1894-1902.

[17] CHUDAK F A, SHMOYS D B. Approximation algorithms for precedence-constrained scheduling problems on parallel machines that Run at different speeds[J]. Journal of Algorithms, 1999, 30(2): 323-343.

[18] HWANG J J, CHOW Y C, ANGER F D, et al. Scheduling precedence graphs in systems with interprocessor communication times[J]. SIAM Journal on Computing, 1989, 18(2): 244-257.

[19] TOPCUOGLU H, HARIRI S, WU M Y. Performance-effective and low-complexity task scheduling for heterogeneous computing[J]. IEEE Transactions on Parallel and Distributed Systems, 2002, 13(3): 260-274.

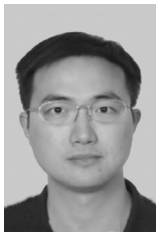
[20] TONG Z, DENG X M, CHEN H J, et al. QL-HEFT: a novel machine learning scheduling scheme base on cloud computing environment[J]. Neural Computing and Applications, 2020, 32(10): 5553-5570.

[21] AHMAD S G, MUNIR E U, NISAR W. A segmented approach for DAG scheduling in heterogeneous environment[C]//Proceedings of the 2011 12th International Conference on Parallel and Distributed Computing, Applications and Technologies. New York: ACM Press, 2011: 362-367.

[22] SCHWIEBERT L, JAYASIMHA D N. Mapping to reduce contention in multiprocessor architectures[C]//Proceedings of Seventh International Parallel Processing Symposium. Piscataway: IEEE Press, 1993: 248-253.

[23] WANG F, XU J, WANG X, et al. Joint offloading and computing optimization in wireless powered mobile-edge computing systems[J]. IEEE Transactions on Wireless Communications, 2017, 17(3): 1784-1797.

[作者简介]



鲁蔚锋（1979− ），男，安徽马鞍山人，博士，南京邮电大学副教授、硕士生导师，主要研究方向为边缘计算、网络安全、区块链等。

李宁（1994− ），男，江苏徐州人，南京邮电大学硕士生，主要研究方向为边缘计算等。

徐佳（1980− ），男，江苏常州人，博士，南京邮电大学教授、博士生导师，主要研究方向为群智感知、边缘计算、无线充电、区块链等。

徐力杰（1983− ），男，江苏盐城人，博士，南京邮电大学副教授、硕士生导师，主要研究方向为传感网/物联网、无线充电网络、边缘计算、移动与分布式计算等。

徐建（1979− ），男，江苏江阴人，博士，南京理工大学教授、博士生导师，主要研究方向为智能运维、网络安全等。